UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS - ICEx
CIÊNCIA DA COMPUTAÇÃO

GABRIEL HENRIQUE SOUTO PIRES

RELATÓRIO DO TRABALHO PRÁTICO
PROBLEMA DO CAIXEIRO VIAJANTE (PCV)

BELO HORIZONTE
2015

GABRIEL HENRIQUE SOUTO PIRES

RELATÓRIO DO TRABALHO PRÁTICO

PROBLEMA DO CAIXEIRO VIAJANTE (PCV)

Trabalho prático apresentado à disciplina de Algoritmos e Estrutura de Dados 2 do curso de Ciência da Computação como requisito parcial para obtenção de nota.

BELO HORIZONTE

2015

Parte I

O problema dado, chamado de Problema do Caixeiro Viajante ou PCV, consiste em achar o melhor caminho a se percorrer em um conjunto de cidades sempre retornando à cidade de origem.

Os casos de teste para o programa foram disponibilizados em forma de arquivo de texto a serem lidos em um formato específico. Cada arquivo contém o número de cidades a serem lidas, assim como as coordenadas x e y de cada cidade e seus respectivos números.

Para ler os arquivos e armazenar os dados lidos de forma mais organizada, foi criado uma $struct$ com os atributos n, x, e y que correspondem respectivamente ao número da cidade, sua coordenada x e sua coordenada y.

Os arquivos são lidos na função $lerCoordenadas$ que recebe como parâmetro um vetor do tipo $Cidade$ e o arquivo a ser lido. Como a linguagem usada no programa foi C++, optei por utilizar a classe $vector$ ao invés de um array convencional.

```
void lerCoordenadas(vector<Cidade>& cidades, FILE *arqEntrada){
	int numCidades;
	string nomeArquivo;
	Cidade temp;

	fscanf(arqEntrada, "%d", &numCidades);

	for(int i=0; i<numCidades; i++){
		fscanf(arqEntrada, "%d", &temp.n);
		fscanf(arqEntrada, "%d", &temp.x);
		fscanf(arqEntrada, "%d", &temp.y);
		cidades.push_back(temp);
	}
	fclose(arqEntrada);
}
```

A função recebe o vetor enviado por referência que é preenchido com as cidades e suas coordenadas. O vetor é enviado por referência, pois ele será utilizado nas outras funções, então é necessário que ele exista na função $main$ para ser enviado como parâmetro nas outras funções do programa.

Para se escolher de qual arquivo txt serão lidos os dados, é necessário que ao executar o programa seja passado um parâmetro com o diretório e o nome do arquivo a ser lido, por exemplo:

./tp1.out 1 data_test/cities_4.txt arquivo_saida.txt

Com o vetor preenchido com os dados lidos do arquivo, foi possível gerar as combinações de todos os ciclos possíveis, ou seja, todas as rotas existentes entre as cidades lidas do arquivo.

Na primeira parte do trabalho foi criado um algoritmo do tipo Backtracking que gera todas as combinações possíveis de rotas tomando como referência as cidades lidas do arquivo de texto.

```
void swap(Cidade *x, Cidade *y){
	Cidade temp;
	temp = *x;
	*x = *y;
	*y = temp;
}
//Parte 1
void permute(Cidade *cidades, int indice, int final, FILE *arqSaida){
	if(indice == final){
		double custoCiclo = somaCusto(cidades, final);
		for(int i=0; i<=final; i++){
			fprintf(arqSaida, "%d ", cidades[i].n);
		}
		fprintf(arqSaida, "%d = $%lf \n", cidades[0].n, custoCiclo);
	}
	else{
		for (int j = indice; j <= final;  j++){
			swap((cidades+indice), (cidades+j));
			permute(cidades, indice+1, final, arqSaida);
			swap((cidades+indice), (cidades+j)); //backtrack
		}
	}
}
```

A função $permute$ recebe um vetor que contém as cidades lidas do arquivo de entrada, assim como a primeira e última posição a se permutar e o arquivo de saída. Esta função gera todas as combinações possíveis com as cidades fornecidas sendo que a primeira cidade foi fixada para facilitar a eliminação de ciclos repetidos, ou seja, a primeira cidade nunca é adicionada na permutação então todos os ciclos sempre começam da cidade 1.

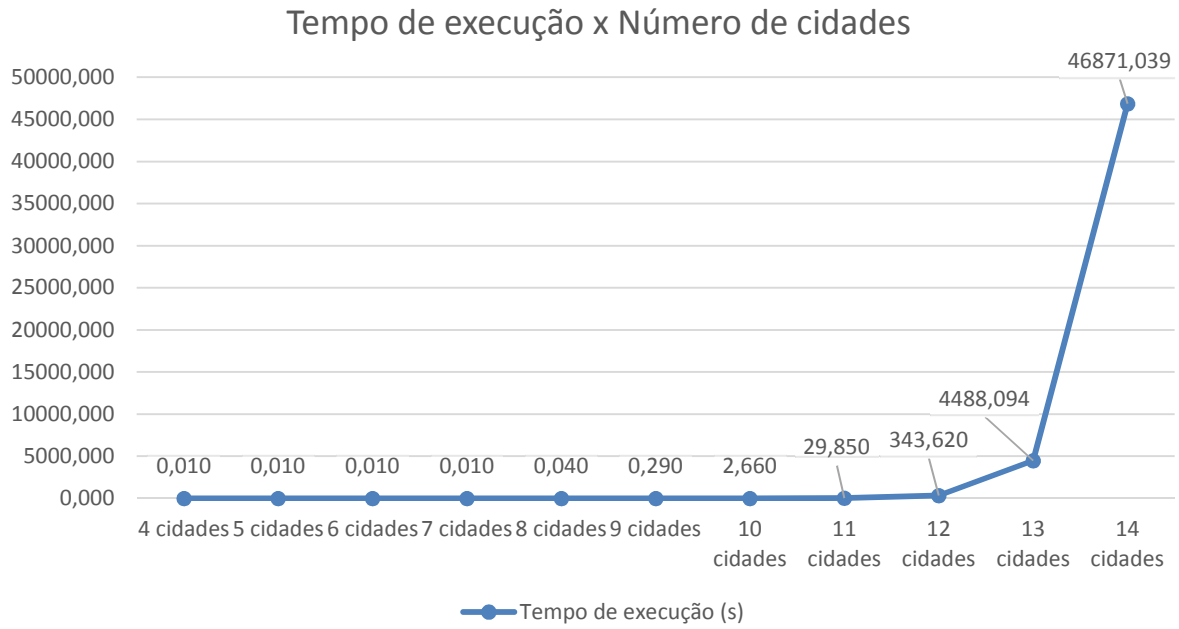
A função é chamada n-1! Vezes dentro de um loop que roda n vezes baseado no último elemento a ser permutado. A partir disso é possível calcular a complexidade assintótica no pior caso desse algoritmo.

Complexidade assintótica: O(n*n!)

Parte II

Dado o conjunto de instâncias (casos de teste), foram testados alguns arquivos para se achar uma solução ótima. Foi notado que quanto maior o número de cidades, mais tempo o programa levava para achar a solução. Pôde ser notado que acima de 7 cidades o tempo gasto de um arquivo para o outro aumentava em cerca de 10 vezes, como mostram a tabela e o gráfico abaixo.

	Número de cidades	**Melhor custo do trajeto**	**Caminho**	**Tempo de execução(s)**
Instância 4	4	113,91	1 2 4 3 1	0.010
Instância 5	5	255,521	1 3 4 2 5 1	0,010
Instância 6	6	249,509	1 2 6 4 5 3 1	0,010
Instância 7	7	198,749	1 2 7 4 6 5 3 1	0,010
Instância 8	8	249,597	1 3 2 8 7 6 4 5 1	0,040
Instância 9	9	265,335	1 5 8 2 6 4 7 3 9 1	0,290
Instância 10	10	315,333	1 6 4 3 7 10 9 2 5 8 1	2,660
Instância 11	11	287,91	1 5 10 7 4 8 3 9 6 2 11 1	29,850
Instância 12	12	317,338	1 4 9 7 3 12 11 5 6 2 8 10 1	343,620
Instância 13	13	378,445	1 7 2 3 10 5 9 12 4 6 13 8 11 1	4488,094
Instância 14	14	299,925	1 3 6 10 2 14 11 13 9 12 5 8 4 7 1	46871,039



Com mais de 14 cidades, a execução desse algoritmo levaria tempo demais para ser viável achar a solução ótima, então só foram feitos os testes até este número. Seria possível criar um algoritmo mais eficiente que calculasse menos distancias, acelerando assim a execução do mesmo, mas um programa assim usaria muita memória RAM e não seria viável executá-lo em qualquer máquina.

Parte III

Na última parte do trabalho, foi solicitado que o PCV fosse solucionado utilizando a Heurística do Vizinho Mais Próximo que consiste em iniciar o ciclo em uma cidade e continuar com a cidade mais próxima ainda não visitada. Para se descobrir o melhor caminho usando essa heurística foi criada a seguinte função:

```
void vizinhoMaisProximo(vector<Cidade> cidades, Cidade* melhorVizinho, ofstream
&arqVizinho){
	Cidade aux;

	for(int i=0; i<cidades.size()-1; i++){
		for(int j=i+1; j<cidades.size(); j++){
			if(dist(cidades[i], cidades[j]) < dist(cidades[i], cidades[i+1])){
				aux = cidades[i+1];
				cidades[i+1] = cidades[j];
				cidades[j] = aux;
			}
		}
	}
	if(somaCusto(&cidades[0], cidades.size()-1) < somaCusto(melhorVizinho,
cidades.size()-1)){
		for(int i=0; i<cidades.size(); i++){
			melhorVizinho[i] = cidades[i];
		}
	}
}
```

A função $vizinhoMaisProximo$ recebe como parâmetro um vetor com as cidades lidas do arquivo, um vetor que guarda o ciclo com menor custo gerado pela função e o arquivo no qual será gravada a saída do programa.

O ciclo do vizinho mais próximo é gerado a partir do vetor original lido do arquivo de entrada. O caminho é testado começando de cada cidade e percorrendo todas as outras, indo sempre para a cidade que tem menor distância da atual. Para achar qual cidade é a mais próxima, é usado um *for* que percorre o vetor nas posições que correspondem às cidade que ainda não foram visitadas, neste *for* é testado uma condição que diz se a distância da cidade atual para a cidade j é menor do que a distância da cidade atual para a próxima cidade no vetor, se for a cidade j é trocada de lugar no vetor com a cidade posterior à cidade atual, ou seja, a cidade j vai para a posição $i + 1$ e a cidade que antes estava na posição $i + 1$ vai para a posição j, o que é equivalente a dizer que o vetor de cidades está sendo ordenado de acordo com a distância entre elas.

Dessa forma é possível percorrer apenas a parte do vetor que corresponde às cidades que ainda não foram visitadas, sem a necessidade de criar uma variável de controle que diz se a cidade já foi ou não visitada. Observe que a variável j sempre recebe $i + 1$ ao entrar no loop interno, o que faz com que as posições à esquerda da cidade atual sejam ignoradas.

Com essa heurística nem sempre é possível achar a solução ótima, ou seja, aquela com o menor custo entre todas as soluções possíveis, mas é possível fazer uma comparação entre a solução ótima

encontrada pelo algoritmo da parte 2 e as soluções encontradas pelo Vizinho Mais Próximo. Essa comparação é feita calculando-se o gap de otimalidade (G) entre a solução ótima S^*e a solução da heurística S:

$$G = \frac{S - S^*}{S^*}$$

	Número de cidades	**Solução ótima**	**Solução da heurística**	**Gap de otimalidade**
Instância 4	4	113,91	113,91	0
Instância 5	5	255,521	255,521	0
Instância 6	6	249,509	249,509	0
Instância 7	7	198,749	198,749	0
Instância 8	8	249,597	249,597	0
Instância 9	9	265,335	294,120	≈ 0,10 = 10%
Instância 10	10	315,333	319,004	≈ 0,01 = 1%
Instância 11	11	287,91	305,709	≈ 0,06 = 6%
Instância 12	12	317,338	328,374	≈ 0,03 = 3%
Instância 13	13	378,445	413,119	≈ 0,09 = 9%
Instância 14	14	299,925	317,882	≈ 0,05 = 5%

A função criada para achar a solução da Heurística do Vizinho Mais Próximo é bastante simples, utilizando apenas 2 loops aninhados para percorrer o vetor de cidades e outro loop separado para atualizar o vetor que guarda o melhor caminho encontrado pela heurística.

Complexidade assintótica: $O(n^2)$